Enhanced Sampling Toolkit Documentation

Release 0.1a

Jeremy O. B. Tempkin

January 06, 2016

CONTENTS

CHAPTER

ONE

INTRODUCTION

The objective of the Enhanced Sampling Toolkit is to provide a programming toolkit that facilitates rapid prototyping and development of enhanced sampling algorithms for use in molecular dynamics applications.

The toolkit is implemented in 100% Python and is targeted for development in the Python language. One reason for this decision is that Python is a popular language in the broader scientific community and has a wide support base in terms of scienfitic code. However, more importantly, a strength of the language lies in the speed at which ideas can be implemented into working code. Since rapid prototyping of new algorithms is the core priority of the toolkit, Python seems a natural choice in language. However, if the need arises in a future date, ports to other languages may be considered and integrated into the package.

In many places of the toolkit some care has been made to adhere to the powerful Python object design principles inherent in the Python data model. We find this to be a powerful feature of the Python language since it facilitates clean, "Pythonic" use of the toolkit in our applications. We are continually working to improve this aspect of the toolkit as the project develops.

In the basic sense, the toolkit serves to wrap commonly used MD codes and in the process abstract the interactions between algorithm level code and the MD engine. This abstractions provides useful extensibilty in the sense that algorthm codes that are implemented in the Walker API can be reused and swapped between MD models and even entire MD codes. Furthermore, the expensive integration steps are executed in faster compiled codes and avoids some of the inefficiencies introduced in the choice of Python.

In addition to rapid algorthm prototyping, we've found in it's development that the toolkit is effective in HPC environments as well. Because the underlying dynamics are executed in commonly used MD codes, the toolkit has access to the HPC features that have been optimized in these codes and can therefore leverage MPI parallelism concurrently at the algorithm level and the MD level as well as support for accelerators such as GPU's or Intel MIC cards.

The structure of the toolkit comprises of two parts. At the core, we provide a specification we call the Walker API. The core API serves to define a set of interactions between algorithm level code and the underlying MD engine. This specification also defines the features developers need to implement for a new dynamics engine to leverage the full features of the toolkit.

On top of this core API, we've implemented several common enhanced sampling algorithms. These tools are built as importable Python modules which can be used to develop new variants of these

CHAPTER

TWO

WALKER API

The central design principle of the Walker API is to provide an abstracted interface between the "algorithm" level code and the code that integrates the dynamics of the model. This section describes the base API used to define the walker objects. The base class is declared internally using Python's abstract base class module. The implementations of the bindings specific to the dynamics packages are registered to this base class definition through direct subclassing of the modules. A key feature of this decision is that it enforces the implementation of the dynamics bindings to implement each of the following methods in their class declarations. However, many of the methods can be overridden with empty methods if an implementation is incomplete or does not require all of the base class methods.

2.1 The Core Walker Application Programming Interface (API)

DEVELOPER NOTE: we may want to use abstract properties in the future.

class walker_base.**walker**

This object acts as an abstract base class that sets the core specification defining the Walker API. These routines

addColvars(*cv*)

Adds a new collective variable to the walker. Subsequent calls to routines that act on the defined collective variables will use the complete list of collective variables in the order they are added. To wipe this list of variables, see the destroyColvars() routine.

close()

Destroys the walker.

destroyColvars()

Removes all collective variables from the walker.

drawVel()

Draws a new value of the velocities for the walker. Can redraw velocities according to a uniform or Gaussian distributions.

equilibrate(*center*, *restraint*, *numSteps*)

getColvars()

This function returns the location of the walker in the collective variable space. Values are returned as a one-dimensional numpy array

getConfig()

This function should return the postition of the walker in configuration space as a one-dimensional numpy array. The coordinates are returned as [x1, y1, z1, x2, y2, z2, ...].

getVel()

This function returns the velocities of the system as a one-dimensional numpy array. The velocities are

returned in a format that matches the getConfig() routine, specifically [v_x1, v_y1, v_z1, v_x2, v_y2, v_z2, ...].

propagate(*numsteps*)
Integrates the dynamics of the model forward in time. Takes the numSteps argument specifying the number of time steps to take.

removeOutput()
This routine adds a source of output for the walker to write information to disk.

reverseVel()
This function reverses the velocity of the walker. The new velocities are given as

setConfig(*configuration*)
Sets the configuration of the At minimum, it should take some sort of specification of the configuration.

setOutput()
This routine adds a source of output for the walker to write information to disk.

setVel()
This routine sets the velocities of the system.

2.1.1 Dynamics Modules

This module contains definitions of the dynamics classes using to wrap information about driving dynamics inside the walkers.

class dynamics.**baoab**(*temperature*, *damping_coefficient*, *shake=False*, *seed=None*, *linear_momentum=True*)
This class implements the baoab integrator.

class dynamics.**langevin**(*temperature*, *damping_coefficient*, *shake=False*, *seed=None*, *linear_momentum=True*)
This class specifies the langevin dynamcis parameters needed.

2.1.2 Setting Walker Output Files

Created on Thu Apr 16 15:03:56 2015

@author: jeremytempkin

class outputClass.**outputClass**(*name*, *outputType*, *filename*, *nSteps=1000*)
This class acts a container object for an output for a walker object.

2.1.3 Setting Walker Collective Variables

A container class for collective variable definitions. We provide here a container class for

2.2 LAMMPS Walker Module

This module implements the Walker API for the LAMMPS MD engine. See walker_base.py for a specification of the API. For details concerning the usage of the LAMMPS MD package see the excellent documentation at the LAMMPS webpage:

http://lammps.sandia.gov/

In particular, you may want to see how the Python wrapper to LAMMPS on which this implementation is based:

http://lammps.sandia.gov/doc/Section_python.html

Here we will outline basic usage guides for the walker API usage in LAMMPS. This module implements the Walker API for the LAMMPS MD engine. See walker_base.py for a specification of the API. For details concerning the usage of the LAMMPS MD package see the excellent documentation at the LAMMPS webpage:

http://lammps.sandia.gov/

In particular, you may want to see how the Python wrapper to LAMMPS on which this implementation is based:

http://lammps.sandia.gov/doc/Section_python.html

Here we will outline basic usage guides for the walker API usage in LAMMPS. To begin using the lammpsWalker module

- highlight here an example code which shows how to use the LAMMPS walker module.

class `lammpsWalker.`**`lammpsWalker`**(*inputFilename*, *logFilename*, *index=0*, *debug=False*)

This class implements the enhanced sampling walker API for the bindings to the LAMMPS package.

Some usage issues to note:

1) Collective variables (CVs) are defined to the walker by constructing a list of CVs internally in the walker.colvars object. These CVs list takes the following format:

["coordinateType", atomids....]

The coordinate type specifies which type of coordinate the CV is (i.e. bond, angle, dihedral, etc.). The next items in the list are the atom indecies involved in this specific instance of the CV.

The walker will use this list to initialize them to the underlying LAMMPS objects.

`addColvars`(*name*, *cvType*, *atomIDs*)

Implements the addition of a collective variable to the list of collective variables held by this walker.

name - A internal string used to reference this collective variable. Collective variable names must be unique.

cvType -

A string refering to the type of collective variable. Currently the following variables are supported:

- 'bond'
- 'angle'
- 'dihedral'
- 'x', 'y', 'z' positions
- 'x', 'y', 'z' velocity components

atomIDs -

A list of the atom indecies involved in the collective variable. Should provide the right number of atom indecies f

- 'bond' -> 2
- 'angle'-> 3
- 'dihedral' -> 4
- position or velocity component -> 1

close()
This function closes the LAMMPS object.

command(*command*)
This function allows the user to issue a LAMMPS command directly to the LAMMPS object.

destroyColvars()
This function removes the colvars set by setColvars(). By default, it removes all of the collective variables in the list. It does not remove them from the collective variables list.

drawVel(*distType='gaussian'*, *temperature=310.0*, *seed=None*)
This function redraws the velocities from a maxwell-boltzmann dist.

equilibrate(*center*, *restraint*, *numSteps*)
This function prepares a LAMMPS image to be at the specified target position given by the vector 'center' passed and an arguments.

getColvars()
This function returns the current position of the LAMMPS simulation in colvars space.

getConfig()
This function returns the current position of the LAMMPS simulation.

getVel()
This function returns the current velocities from the LAMMPS simulation.

minimize(*args=None*)
This function runs a minimization routine with the specified type.

propagate(*numSteps*, *pre='no'*, *post='no'*)
This function issues a run command to the underlying dynamics to propagate the dynamics a given number of steps.

removeOutput()
This routine removes the ouput pipes for information to be written to disk from the underlying dynamics engine. Right now this simply clears all existing output.

reverseVel()
This function reverses the velocities of a given LAMMPS simulation

setConfig(*config*)
This routine sets the internal configuration.

setDynamics(*dynamics_instance*)
This routine sets the dynamics for the walker.

setOutput(*name*, *outputType*, *filename*, *nSteps*)
This routine sets up a mechanism for writing system information to file directly from the dynamics engine. This is equivalent to output constructed

setTemperature(*temp*)
This function sets the temperature of the walker object.

setTimestep(*timestep*)
This routine sets the dynamics time step.

setVel(*vel*)
This function sets the velocity to the lammps simulation.

2.3 OpenMM Walker Module

We plan to implement an OpenMM walker API module in a future release.

CHAPTER

THREE

APPLICATIONS

3.1 Nonequilibrium Umbrella Sampling

3.1.1 Partition Module

This module contains the definition of the partition object. The partition object represents the definition of a spatial decomposition defined in umbrella samping / stratification algorithms.

The basic features of this partition and how one uses this data structure to perform umbrella sampling type calculations is described as follows.

class `partition.`**`partition`**(*scratchdir=None*, *parallel=False*)

This class defines a set of windows that partition the sampling space. This class will contain an array of basisFunction objects.

`accumulateObservables`(*wlkr*, *sample*, *colvars*, *indx*)

This routine loops through the observables list and updates the samples in the corresponding windows.

`addObservable`(*A*, *rank_index=None*)

This routine adds an observable and initializes local copies of the observables in the basis windows.

`build_keylist_to_index_map`(*keylist*)

This routine takes the input keylist and constructs internal dictionaries that convert between indicies of F and elements of the keylist.

`build_neighbor_list`(*umbrellas*, *s*, *debug=False*)

This routine constructs a neighborlist based on the radius of the basis function.

L is the vector specifying the periodic lengths in each dimension.

`communicateMPI`(*rank*, *comm*, *sparseSolve=False*, *finiteTime=False*, *debug=False*)

This routine performs a round of MPI communication to synchronize information across all ranks.

Let's note here that there is a sequence of communication / computation that takes place in this section. Therefore it makes the most sense to interleave the two components here.

It should be noted that this being a initial version of the parallel neus code, an optimization of the MPI communication could be implemented at a later date.

`computeObservables`(*rank=None*)

This routine populates the partition observables with data averaged from the windows.

`computeZ`(*sparseSolve=True*, *finiteTime=False*)

Solves for z vector given current G,a via solving the following linear system:

(I - G)^T z = a

for a finite time process and

zG = z

for infinite time process.

A = (np.identity(self.G.shape[0]) - self.G).transpose()

self.z = np.linalg.solve(A, self.a)

get_basis_function_values (*wlkr*, *umbrella_index=None*)
This function takes a point in collective variable space and returns an array of the value of the basis functions at that point.

If no umbrella index is passed, then search the whole space

reinject (*wlkr*, *i*)
This function initializes a simulation from the entry point list in the current umbrella.

removeObservable ()
This routine removes all observables from the list for the partition.

resetObservable (*obs*)
Set each element in the data of the passed observable to zero.

sample (*wlkr*, *umbrellaIndex*, *numSteps*, *stepLength*, *walkerIndex*, *corrLength=None*, *debug=False*)
This function takes a system lmp and propagates the dynamics to generate the required samples but storing and generating samples via NEUS algorithm.

Specifically, this performs an umbrella sampling routine using NEUS reinjection procedure for reinitializing the walker.

We should remove the need for the output to be specified internally here.

set_umbrellas (*umbrellas*, *neighborList=True*, *s=None*)
This routine replaces the current list of umbrellas with those provided and updates the matricies contained to match the new umbrella list. This behavior is destructive to the old matricies.

updateF (*row*, *epsilon*)
This routine update G according to:

G_{ij}^{k+1} = (1 - epsilon_{k}) * G_{ij}^{k} + epsilon_{k} * M_{ij} / T_{i}

for a finite time problem.

3.1.2 Basis Function Module

Created on Mon May 5 18:04:54 2014

@author: jtempkin

class basisFunctions.**Box** (*center*, *width*, *periodicLength=None*, *max_entrypoints=250*)
This class impements a rectangle in CV space.

indicator (*coord*)
Return the value of the indicator of the box at this point. This will be 1.0 if coord is contained inside the box, and 0.0 if it is not.

class basisFunctions.**Gaussian** (*mu*, *sig*, *max_entrypoints=250*)
This class implements a Gaussian function.

indicator (*coord*)
Function that returns the value of the Gaussian at a point.

class basisFunctions.**Pyramid** (*center*, *width*, *ref_center=None*, *ref_width=None*, *time=None*, *periodicLength=None*, *max_entrypoints=500*)

This class implements a pyramidal basis function.

center The center of the box in the collective variable space. This should ideally be a numpy/scipy array or list, but other iterables MIGHT work.

width This is the width out from the center in each collective coordinate. The actual width of the box in each coordinate is twice this vector.

periodicLength The wrapping object takes a little explanation. It is an optional array, where the i'th element corresponds to the i'th collective variable. If the collective variable wraps around, the corresponding element is the range of the collective variable, e.g. 360 for an angle going from -180 to 180. If the collective varable does not wrap around, the corresponding element is just 0. If none of the collective variables wrap, leaving wrapping as None is perfectly fine.

ref_indicator(*coord*)
Return the value of the indicator for the reference coordinates if appropriate.

NOTE: Currently we will simply implement the reference discretization as non-overlapping boxes.

time_indicator(*coord*)
Return the value of the indicator function for the time coordinate.

This will be implemented currently as non-overlapping discretization in time.

class basisFunctions.**basisFunction**(*center*, *width*, *ref_center=None*, *ref_width=None*, *time=None*, *periodicLength=None*, *max_entrypoints=500*)
It is important to note that the configurations associated with sampling of this box are stored at this level of the class structure.

At this level, the characteristics defining the basisFunction class is a set of configurations inherent to the class.

add_entry_point(*ep*, *key*)
This routine adds a new entry point to the current list

add_local_obserbale(*obs*)
This routine adds a local observable to this window.

add_new_entry_point(*ep*, *key_from*)
This routine add an entry point to the list of proposed entry points.

empty_new_entry_points()
This routine empties the new entry points data structure locally.

flush_data_to_file(*filename*)
This function flushes the internal data buffers configs and samples to files.

This function uses the HDF5 python implementation to store data to disk. It should be noted that it creates two groups in the top-level called 'colvars' and 'timeSeries'. Because appending data is not as easy for this file format, it stores each flush as a new dataset indexed in each group.

get_entry_point()
This routine returns an entry point from the buffers draw proportional to the entry fluxes.

get_local_observales()
This routine returns a list of the local observables.

initialize_entry_points(*keylist=None*)
This routine initializes the entry point data structure in the umbrella.

The structure of the entry Point library is designed to be flexible as to how one wants to track entries from neightbors. There are two schemes that are supported now. The first option is a single set containing all entry points to this window (we're actually using the Python set structure). The second type is a dictionary of sets. This allows one to group contributions to the entry point list for this window by assigning a key to

each neighbor and add/draw entry points from these separate sets by passing a key value to the respective add/draw operations.

To construct the second type of structure you specify the key structure with a list of keys that define the categories from which one can group entry points.

remove_local_observales()
This routine removes all local observables contained by this window.

update_entry_points_fluxes(*flux*, *key_map*)
This routine updates the sizes of the discretization of the entry point flux lists based on the maximum size of the entry point buffer.

The flush keyword moves all stored points to the active buffer regardless of the fluxes

3.1.3 Entry Point Module

This module acts as a template for defining named tuples that store entry points.

It provides a consistent entry point contruction as a named tuple.

class entryPoints.**entry_point**(*q*, *p*, *ref_q*, *ref_p*, *time*, *cv*)
class entryPoints(object):

This class acts as a wrapper for an entry point object. Should have at least the structures to contain a phase space point.

def __init__(self, config, vel, time):

Constructs an object containing at least a point in phase space. Can be initialized to contain other information like a reference point as well but we leave this to be handled dynamically.

self.config = config self.vel = vel self.time = time

cv
Alias for field number 5

p
Alias for field number 1

q
Alias for field number 0

ref_p
Alias for field number 3

ref_q
Alias for field number 2

time
Alias for field number 4

3.1.4 Observable Module

This module contains a list of observable routines. These functions serve to calculation and record the value of observables in the course of a calculation.

These routines are initialized as their own class and subsequently can be invoked during other calculations or called by other routines. We may consider updating these classes with a base class prototype similar to how the walker class was initialized.

class observables.**P1**(*name*, *s*, *stepLength*, *atomids*, *data*, *cellDim*)
This routine returns the TCF of the end to end distance.

class observables.**cv_indicator_correlation**(*name*, *s*, *stepLength*, *data*, *cv_index*, *cv_range*)
This routine returns the value of an indicator function over a space in a given cv.

class observables.**dihedral_fluctuation_correlation**(*name*, *s*, *stepLength*, *data*, *cvindex*, *mean*)
This routine returns the TCF of a dihedral angle.

class observables.**dihedral_fluctuation_correlation_2**(*name*, *s*, *stepLength*, *data*, *atomids*, *mean*)
This routine returns the TCF of a dihedral angle.

class observables.**dist_fluctuation_correlation**(*name*, *s*, *stepLength*, *atomids*, *data*, *cellDim*, *mean*)
This routine returns the TCF of the end to end distance.

class observables.**electric_field**(*name*, *s*, *stepLength*, *atomids*, *data*, *cellDim*, *atom_exclusions*, *ref_atoms_ids=None*)
This class constructs an observable that reports on the time-correlation of the electric field at a point in space.

class observables.**pmf**(*name*, *data*, *data_width*)
This class represents a PMF observable.

3.2 Steered Transition Path Sampling (Steps)

3.3 Replica Exchange

3.4 Profiling Applications

CHAPTER

FOUR

TESTING

Here I will add a description of the unit testing of Walker API implementations and potentially the application testing.

A

B

C

D

E

F

G

I

L

M

O

P

Q

R

S

T

U

W